COMP 424 – Project Report

Ismail Faruk – 260663521

pentagon-swap

Motivation

In most game playing such as pentagon-swap, a typical approach would be implementing a minimax algorithm. Upon some research I discovered an algorithm called Negamax, which is a variant form of minimax search that relies on the zero-sum property of a two-player game. My main motivation for solving pentagon-swap was heavily based on a research paper by Buescher Niklas [2] and generic implementations of tic-tac-toe. Buescher's paper is based on the game pentago-rotate, and not pentago-swap. But the concept of implementing a search tree was what led me to study his work. In his paper, Buescher refers to an open source artificial intelligence named *pentagod* by Tewalds. Tewalds created his pentagod using a negamax function and he claims, "As far as I know, the negamax version is the strongest pentago player in existence right now." I initially tried using minimax but then I read up on negamax. I appreciated the implementation more and hence, I ended up implementing something like Tewalds' *pentagod*.

I implemented a Negamax alpha-beta pruning algorithm, with iterative depth search, an optimum move search and a simple utility function. The negamax algorithm would create a search tree, prune it and return the value of the best node and the optimum search algorithm would compare it with previous values and return the best moves possible. The utility function would evaluate if the move was a win, loss or draw. This implementation was able to beat the random player on many tries, but not all. Thus, it is a soft-solved solution.

Brief Description

Negamax is in principal an Alpha-Beta algorithm which combines the two different functions for the MAX and the MIN node to maximize the outcome for both players. Therefor only one function is called with inverted alpha and beta values for opposite players. More precisely, the value of a position to P1 in such a game is the negation of the value to P2. My negamax function looks for a move that maximizes the negation of the value resulting from the opponent's move. A single procedure can be used to value both positions. This implementation was a simplification of minimax, which would require that P1 selects the move with the maximum-valued successor while P2 selects the move with the minimum-valued successor. The algorithm relies on the property that **Max** (alpha, beta) = −**MIN** (−beta, −alpha) to simplify the implementation of the minimax algorithm, i.e. the Max player's alpha beta value is negative to that of the MIN player.

Negamax sets the arguments α and β for the root node to the lowest and highest values possible. When negamax encounters a child node value outside an alpha/beta range, the negamax search cuts off thereby pruning portions of the game tree from exploration. Cut offs are implicit based on the node return value. Because, such values would up be generating extra nodes and can be disregarded when trying to get negamax value at its root node. Alpha beta pruning attempts to guess the most probable child nodes that would yield the node's score, therefore pruning additional unbeneficial and/or repeated negamax tree branches and remaining child nodes from the search tree.

In the end, after the depth is searched, I evaluate the result using the utility function and update my move be comparing it with the local maxima.

Summary of approach

The first reason to use negamax was that it works. It combines than typical minimax with alpha-beta pruning into one function, where is checks both alpha and beta values in comparison to the node. This is a better process than only checking alpha or beta once based on the player. It avoids extra branching, which is done by alpha-beta minimax, by calling itself and exchaging the alpha and beta values. Furthermore, the evaluation function outputs higher values if an advantage for the actual player is calculated and lower or negative values for a disadvantage. In contrast, the former evaluation function returned turn independent high values if an advantage for the MAX player existed.

The disadvantage was that I could not search more nodes, which increased the chances of failure. A player with a better utility function that had better heuristics would be able to beat this implementation as it lacks a strong utility function and is depth limited.

My implementation was not able to win all matches with the random player. This was partly because of the depth limitation, but mainly due to the lack of detailed heuristic in the utility function. Thus, in the initial phase of the game, there could be high chances of worse moves being made, making it lose to the random player.

Other methods

Another approach I tried was a generic minimax function with alpha-beta pruning, with the same optimum search and utility function. This allowed me to search 1 more depth. But the reason I did not implement it was because only one of the node values would only updated based on the implementation, but I wanted to compare the value of the move with alpha and beta at the same

time and thus implemented negamax. My implementation was able to win more matches to the alpha-beta pruning I had done, thus it was a soft-solved implementation, than a hard-solved.

I also tried to implement the transposition table with the negamax function, but I could not finish it, thus added information about it to the improvement section. It would function better than the negamax function I had implemented, no doubt.

Improvement

I would improve the utility function in detail. As it is implemented, the utility function only considers win, loss and draw, but that cannot be always reached with 2 moves and it is not a good evaluation function. It is a soft-soled implementation at best. A better approach would be to implement a utility function based on the configuration of the board. The utility function would evaluate the board state and return the utility value based on the degree of advantage the player has on the board and vice-versa. This would allow comparison of states more clearly and the optimum move search algorithm would be able to find something closer to global optimum than it has now. Moreover, this would help evaluate the initial moves more precisely.

I would order the negamax search tree prior to each loop that evaluates child nodes. It would be an optimization to the for alpha beta pruning, as it attempts to guess the most probable child nodes that would yield the node's score. The algorithm searches those child nodes first. The result being the desired value appear earlier and more frequent alpha/beta cut offs occur, therefore pruning additional unbeneficial and/or repeated negamax tree branches and remaining child nodes from the search tree. This procedure might be cost intensive but as shown earlier a good move ordering may half the search depth.

I would implement negamax with alpha beta pruning and transposition tables. The transposition tables would selectively store the values of nodes in the game tree. When negamax would search the game tree, and encounter the same node multiple times, a transposition table can return a

previously computed value of the node, skipping potentially lengthy and duplicate re-computation of the node's value. I would store the transposition table into a file and utilize the first 30 seconds to retrieve the table and select the first move and subsequent move based on it.

References

1. 2019. *Negamax.* February 3 . https://en.wikipedia.org/wiki/Negamax.
2. Niklas, Buescher. 2011. "On Solving Pentago." Fachbereich Informatik, Technische Universität Darmstadt, Darmstadt.